Naval Research Laboratory

Washington, DC 20375-5320

NRL/MR/5540--07-9093

Documenting Xenon's Page_Alloc Module

James Kirby, Jr.
John McDermott
Myong Kang
Bruce Montrose

Center for High Assurance Computer Systems
Information Technology Division

December 10, 2007

REPORT DOCUMENTATION PAGE		*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* 10-12-2007	**2. REPORT TYPE** Memorandum Report	**3. DATES COVERED** *(From - To)* Jan. 2007 – Oct. 2007

4. TITLE AND SUBTITLE Documenting Xenon's Page_Alloc Module	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) James Kirby, Jr., John McDermott, Myong Kang, and Bruce Montrose	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER 6475
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Research Laboratory, Code 5540 4555 Overlook Avenue, SW Washington, DC 20375-5320	**8. PERFORMING ORGANIZATION REPORT NUMBER** NRL/MR/5540--07-9093
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)	**10. SPONSOR / MONITOR'S ACRONYM(S)**
	11. SPONSOR / MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

One of the critical assurance requirements for achieving medium or high assurance is a requirement for significant modularity in design and implementation. As part of the Xenon effort to create a secure Xen with a medium to high degree of assurance, we have embarked on its remodularization, a documented decomposition into well-defined pieces with well-defined relationships among them. This remodularization of Xen is based on the information hiding principle. Associated with an information hiding module may be a provided interface, a set of public programs (e.g., functions, subroutines, macros) that programs outside the module can use to accomplish their work. Documentation of a module's provided interface serves as a contract between the module's users and its developers. This report documents the decomposition of the Xen page_alloc module and the specification of the provided interface of each of its submodules.

15. SUBJECT TERMS

Secure software engineering	Information hiding design	Xen hypervisor
Software modularity	Software interface specification	

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** James Kirby, Jr.
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**	UL	37	**19b. TELEPHONE NUMBER** *(include area code)* (202) 767-3107
Unclassified	Unclassified	Unclassified			

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

CONTENTS

Documenting Xenon's Page_Alloc Module*

J. Kirby, J. McDermott, M. Kang, and B. Montrose
Naval Research Laboratory

December 7, 2007

Abstract

One of the critical assurance requirements for achieving medium or high assurance is a requirement for significant modularity in design and implementation. As part of the Xenon effort to create a secure Xen with a medium to high degree of assurance, we have embarked on its remodularization, a documented decomposition into well-defined pieces with well-defined relationships among them. This remodularization of Xen is based on the *information hiding principle*. Associated with an information hiding module may be a *provided interface*, a set of public programs (e.g., functions, subroutines, macros) that programs outside the module can use to accomplish their work. Documentation of a modules provided interface serves as a contract between the modules users and its developers. This report documents the decomposition of the Xen page_alloc module and the specification of the provided interface of each of its submodules.

Introduction

As part of the Xenon effort to create a secure Xen with a medium to high degree of assurance, we have embarked on its remodularization—a documented decomposition into well-defined pieces with well-defined relationships among them. When the modularization is complete, its documentation will support certification reviews and will help developers and maintainers identify parts of the Xen they must understand to accomplish some task without looking at irrelevant parts.

One of the critical assurance requirements for achieving medium (EAL 5) or high (EAL 6/7) assurance is a requirement for significant modularity in design and implementation. This increased modularity serves multiple purposes. First, increased modularity makes any security analysis more believable. Second, it reduces the scope of a flaw (or malware in some cases); that is modularity reduces dependencies between parts of the software, so flaws remaining in the code are less likely to be exploitable. Finally, modularity can be used to separate code into security-relevant and security-irrelevant modules. This reduces the amount of code that needs to be built according to high assurance rules.

This remodularization of Xen is based on the *information hiding principle*, which Dave Parnas described in his well-known paper, *On the Criteria to Be Used in Decomposing Systems into Modules* [CACM 1972]. A later paper, *The Modular Structure of Complex Systems* [Parnas, Clements, and Weiss, IEEE TSE, March 1985], which reported results of an NRL project to redevelop the operational flight program (OFP) for the Navy's A-7E attack aircraft, describes techniques that aid in applying the principle to a real system.

*This software is a Research Work of the United States Naval Research Laboratory, derived from GPL software. Any distribution of a source code or binary form of this software is prohibited. Release of this software outside the Department of Defense may be a violation of U.S. Law. The derived portion of this software is United States Government Work not protected by U.S. Copyright.

Manuscript approved October 29, 2007.

An *information hiding module* is a design construct. Each module has a *secret*, one or more decisions (which might also be thought of as assumptions) that developers judge likely to change or that they judge it is useful not to distribute throughout the system. Associated with a module may be a set of a public programs (e.g., functions, subroutines, macros) that programs outside the module can use to accomplish their work. These public programs constitute a public or *provided interface* representing decisions and assumptions upon which using programs may depend. When the module is carefully designed, the decisions it hides can be changed without invalidating the decisions and assumptions that the provided interface represents and upon which using programs depend.

A module's secret is decomposed by its children. For example, a module that hides characteristics of peripheral devices that are likely to change might be decomposed into a set of modules, each of which hides characteristics of a particular class of device that are likely to change. This *module structure* or *information hiding structure* for a system is a tree of modules. It is useful to think of each module in the hierarchy as a work assignment for one or more developers or maintainers. The most difficult (and important) parts of designing the module structure are identifying the module secrets and clearly and concisely describing them. As a tree, the module structure can be usefully presented in a variety of ways, including as an indented list and as a UML class diagram.

Documentation of a module's provided interface serves as a contract between the module's users and its developers. To be useful, this documentation should be written so that it does not reveal or assume decisions designers intend the module to hide. Some implications of this are that documentation of the behavior of functions on the provided interface should not be written in terms of such implementation details as their algorithms, internal data structures, nor which functions they may call.

To facilitate describing the behavior of functions on modules' provided interfaces, we develop an *environmental model* which provides an application-specific ontology for the system [Kirby, COMPSAC 2006]. The model records the system boundary by identifying objects in the environment of the system (which may include the system itself and components of the system) and attributes of the objects that may be relevant to the system, referred to as *environmental attributes*. The declaration of an attribute in the model includes its type, which characterizes the values it can assume, and a description of how to interpret its value. Identifying appropriate and relevant attributes—those that describe what the software can sense, control, and affect—is key. Descriptions of software behavior can be written in terms of these attributes.

UML classes represent objects in the system environment. Standard UML class notation may record relationships among the classes of the environmental model, the relative cardinality of the objects abstracted by the classes of the environmental model, and the cardinality of the attributes of each object. The attributes associated with each object are listed in the corresponding class. Attributes whose values the software can sense (either directly or via physical or cyber sensors) are referred to as *monitored attributes*. Attributes whose values the software can set or affect (either directly or via physical or cyber actuators) are referred to as *controlled attributes*. Monitored attributes and controlled attributes can be distinguished by assigning the former to a class compartment labeled *monitored*, and assigning the latter to a class compartment labeled *controlled* (see Fig. 1). Assigning an attribute to an unnamed compartment indicates that the engineer has not decided whether the attribute is monitored or controlled.

Fig. 1. illustrates an environmental model of hardware memory which consists of a number of hardware pages. The monitored attribute mMaxPage gives the number of pages in memory. The monitored attribute mPageSize gives the size of a page in bytes (in this model all pages have the same size). The figure illustrates attributes of a HardwarePage, *e.g.,* monitored attribute mBad, controlled attributes cAllocated, cZeroized. Xen can sense the values of monitored attributes and set the values of controlled attributes. The tabular declarations of attributes in Tables 1 through 3 include a description of how to interpret attribute values.

Section 2.3 illustrates the specification of the function `map_alloc()`, which allocates hardware pages. As indicated by the table, the function has two parameters. Both parameters are inputs to the function (indicated by the *I* in the *Mode* column) and are of type `unsigned long`. The first parameter (*p1*) gives the linear address of a hardware page. The second parameter specifies a number of hardware pages. Below the table, *Undesired Events* identifies undesired events—requesting pages that `map_alloc()` has already allocated and requesting pages before initializing the module *Page Allocator*—which, encountered at run-time, prevent correct operation of the function. *Effects* describes the effect of calling `map_alloc()`, which is to set to *true* the *cAllocated* attribute of all hardware pages with linear addresses in the range

$$[p1, p1 + p2 - 1]$$

Section 2.2 illustrates the specification of the function `allocated_in_map`, which callers can use to determine whether a particular hardware page is allocated. In the parameter table, labeling the first parameter *p0*, rather than *p1*, indicates that it specifies the value returned by the function, which the term *tAllocated* specifies. The *O* in the *Mode* column indicates that the parameter is output from the function to its caller (as would be expected of the function's return value). The definition of *tAllocated* in the *Dictionary* specifies that the value returned by the function (*p0*) is the value of the *cAllocated* attribute of the hardware page whose linear address is given by *p1*.

Some tables in the *Effects* and *Dictionary* subsections of Sections 6.2, 6.3, and 6.4 are more complicated than those discussed above. For example, the first table in *Effects* of Section 6.2 specifies the values on return to the caller of the variables indicated in the leftmost cell below the double line (e.g., *p.cPageOwner*, *p,cDomAllocated*, *p.cRefCount*). The prime appended to the variable name indicates the value of the variables on return. The values of unprimed variables are those established on the call to the function. The cells in the last rows (below the double line) to the right of the double line specify alternative value(s) for the variable(s). The leftmost cells above the double line partition the state space of the function. Exactly one of them is true, which selects the corresponding row of rules for determining the value of the variable(s). These rules are written so that they also partition the state space—exactly one of them is true. If $p1.mDying$ is true—i.e., the domain referred to by the first parameter is dying—then the first row determines which of the alternative set of values in the last row apply. The $true$ in the first column to the right of the double line—which can be thought of as specifying $always$—indicates that the values in the last row to the immediate right of the double line apply. The $false$ in the rightmost column, which can be thought of as $never$, indicates that the values in the rightmost column of the last row do not apply when $p1.mDying$ is true. When $p1.mDying$ is false, the more complex expressions to the right of the double line in the second row determine which set of values in the bottom row apply. Note that the table is quantified by the expression above the table.

The first table in *Effects* in Section 6.3 which has only the vertical line is interpreted differently. The cells to the left of the double line represent alternative conditions. If one of them is true, then the corresponding expression to the right of the double line describes effects of calling the program. If none of the alternative conditions to the right of the double line is true, then the table does not describe any effects of the calling the program.

Module page_alloc

Secret. The page_alloc module's secret is how memory is allocated in Xenon. This includes how Xen keeps track of which pages of memory have been allocated and which have not.

1 Environmental Model of Hardware Memory

Fig. 1 provides a graphical view of Xenon's environmental model of the memory hardware on which it runs. This is a software view of the memory hardware. While the hardware has its own particular addressing scheme based on address lines, this Xenon model uses the linear address scheme. Hardware memory comprises a set of hardware pages. The figure illustrates two attributes of hardware memory, mPageSize and mMaxPage. Being in the *monitored* compartment of the Hardware Memory class indicates that Xenon is able to determine the values of the two attributes, but is unable to change those values.

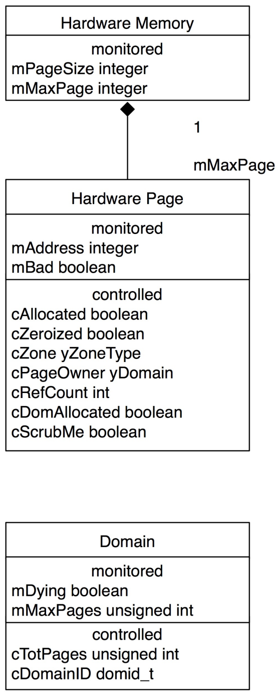

Figure 1: Environmental Model of Hardware Memory

The Hardware Page class in Fig. 1 indicates that hardware pages have attributes whose values Xenon can sense but not change and that it has attributes whose values Xenon can change (attributes in the *controlled* compartment of the Hardware Page class).

Table 1 declares the attributes of hardware memory which the environmental model in Fig. 1 introduced. From Table 1 we see that mMaxPage denotes the number of pages in hardware memory and that mPageSize denotes the size of hardware pages in bytes. Table 3 declares the attributes of hardware pages which the environmental model introduced.

Table 1: Hardware Memory Attribute Declarations

Attribute	**Type**	**Class**	**Interpretation**
mMaxPage	integer	monitored	Denotes the number of pages in hardware memory.
mPageSize	integer	monitored	Denotes the number of bytes in a hardware page.

Table 2: Hardware Page Attribute Declarations

Attribute	**Type**	**Class**	**Interpretation**
mAddress	integer	monitored	Denotes the *linear address* of the hardware page.
cAllocated	boolean	controlled	cAllocated = true iff Xen has allocated the hardware page.
mBad	boolean	monitored	mBad = true iff the hardware page is not to be used.
cZeroized	boolean	controlled	cZeroized = true iff the hardware page is zeroized.
cZone	yZoneType	controlled	Denotes zone to which Xen has assigned the hardware page.
cPageOwner	yDomain	controlled	Denotes a domain to which Xen has assigned the hardware page.
cRefCount	int	controlled	Count of references to the page.
cDomAllocated	boolean	controlled	Indicates whether Xen has assigned the hardware page to a domain.
cScrubMe	boolean	controlled	Indicates a page that Xen needs to zeroize.

Dictionary

yDomain is a handle for a Xen domain.

yZoneType denotes memory zones. Enumerated values are: *xen*, *dom, dma*, *any* (Xen makes limited and inconsistent use of the latter).

mfn2Page(), **page2Mfn()** are functions on addresses.

$$mfn2Page(\text{linear address}) \rightarrow \text{virtual address}$$

$$page2Mfn(\text{virtual address}) \rightarrow \text{linear address}$$

$$(\forall p \in HardwareMemory)(\exists \text{ virtual address } v)(v = mfn2Page(p) \Rightarrow p = page2Mfn(v))$$

$$(\forall \text{ virtual address } v)(\exists p \in HardwareMemory)(p = page2Mfn(v) \Rightarrow v = mfn2Page(p))$$

Fig. 2 graphically illustrates the module structure of the Page_Alloc module. The remainder of this document describes each of the submodules in turn, describing its secret and specifying the programs on its provided interface.

Table 3: Domain Attribute Declarations

Attribute	**Type**	**Class**	**Interpretation**
mDying	boolean	monitored	Indicates whether the domain is dying.
mMaxPages	unsigned int	monitored	Indicates the maximum number of pages that Xen assigns to a domain.
cTotPages	unsigned int	controlled	Indicates the number of pages that Xen has assigned to a domain.
cDomainID	domid_t	controlled	Indicates the identifier of a domain.

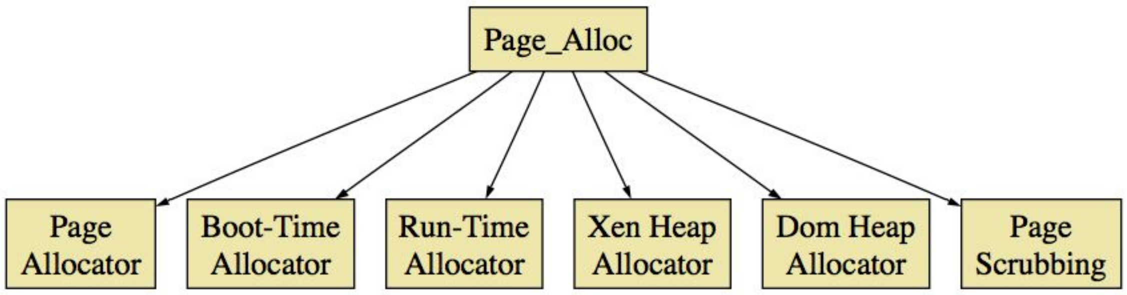

Figure 2: Page_Alloc Module Structure

2 Page Allocator (was *Allocation Bitmap*)

Secret. This module hides how to keep track of which hardware pages of memory have and have not been allocated.

2.1 init_boot_allocator

Initialize boot-time memory allocation mechanism.

Parameter #	Mode	Type	Interpretation
p0	O	paddr_t	Starting memory location of *available* hardware memory to manage.
p1	I	paddr_t	Starting memory location of hardware memory to manage.

Undesired Events

- **uAllocationAlreadyInitialized.** Hardware memory allocation already initialized.
- **uAllocationNotInitialized.** Hardware memory allocation not initialized.

Effects

$$(\forall p \in HardwareMemory)(p.mAddress \text{ in } [p1, mMaxPage - 1] \Rightarrow p.cAllocated' = true)$$

- enables uAllocationAlreadyInitialized
- disables uAllocationNotInitialized

Issues

- Don't think behavior is quite right. Don't think the pages containing the bit map, which is at the beginning of the memory pointed to by *p1*, is allocated.

2.2 allocated_in_map

Hardware page already allocated?

Parameter #	Mode	Type	Interpretation
p0	O	boolean	*tAllocated*
p1	I	unsigned long	Linear address of a hardware page.

Undesired Events

- uAllocationNotInitialized
- uNotLegalAddress

Effects

None.

Dictionary

tAllocated boolean

$$(\exists p \in HardwareMemory)(p.mAddress = p1 \Rightarrow tAllocated' = p.cAllocated)$$

Issues

- Does not report undesired events encountered.

2.3 map_alloc

Allocate hardware pages.

Parameter #	Mode	Type	Interpretation
p1	I	unsigned long	Linear address of a hardware page.
p2	I	unsigned long	Count of hardware pages.

Undesired Events

- **uHardwarePagesAlreadyAllocated.** The requested hardware pages are already allocated.
- uAllocationNotInitialized

Effects

$(\forall p \in HardwareMemory)(p.mAddress \text{ in } [p1, p1 + p2 - 1] \Rightarrow p.cAllocated' = true)$

- Enables the undesired event **uHardwarePagesAlreadyAllocated** for *p2* hardware pages starting at page number *p1*.
- Disables the undesired event **uHardwarePagesNotAllocated** for *p2* hardware pages starting at page number *p1*.

Issues

- Does not report undersired events encountered.

2.4 map_free

Return allocated hardware pages to free store.

Parameter #	Mode	Type	Interpretation
p1	I	unsigned long	Hardware page number.
p2	I	unsigned long	Count of hardware pages.

Undesired Events

- **uHardwarePagesNotAllocated.** Returned hardware pages were not allocated.
- uAllocationNotInitialized

Effects

$$((\forall p \in HardwareMemory)(p.mAddress \text{ in } [p1, p1 + p2 - 1] \Rightarrow p.cAllocated' = false)$$

- Disables the undesired event **uHardwarePagesAlreadyAllocated** for *p2* hardware pages starting at page number *p1*.
- Enables the undesired event **uHardwarePagesNotAllocated** for *p2* hardware pages starting at page number *p1*.

Issues

- Does not report undesired events encountered.

3 Boot-Time Allocator

The Boot-Time Allocator module hides how the initial allocation of memory is performed.

3.1 init_boot_pages

Initial allocation of pages.

Parameter #	**Mode**	**Type**	**Interpretation**
p1	I	paddr_t	Linear address of a hardware page.
p2	I	paddr_t	Linear address of a hardware page.

Effects

$$(\forall p \in HardwareMemory)(p.mAddress \text{ in } [p1, p2] \land p.mBad = true \Rightarrow p.cAllocated' = true)$$

Issues

- Why are there both init_boot_pages and init_boot_allocator? Why not combine?
- What if
$$p1 \geq p2?$$

3.2 alloc_boot_pages_at

Allocate specified number of free pages starting at specified linear address.

Parameter #	Mode	Type	Interpretation
p0	O	unsigned long	*tFirstAllocatedPage*
p1	I	unsigned long	Number of hardware pages.
p2	I	unsigned long	Linear address of a hardware page.

Effects

$$(\forall p \in HardwareMemory)(p.mAddress \text{ in } [p2, p1 + p2 - 1] \Rightarrow p.cAllocated = false) \Rightarrow$$
$$(\forall p \in HardwareMemory)(p.mAddress \text{ in } [p2, p1 + p2 - 1] \Rightarrow p.cAllocated' = true)$$

Dictionary
tFirstAllocatedPage unsigned long

$$(\forall p \in HardwareMemory)(p.mAddress \text{ in } [p2, p1 + p2 - 1] \Rightarrow p.cAllocated = false) \Rightarrow$$
$$tFirstAllocatedPage' = p2$$

$$(\exists p \in HardwareMemory)(p.mAddress \text{ in } [p2, p1 + p2 - 1] \wedge HardwarePage[i].cAllocated = true) \Rightarrow$$
$$tFirstAllocatedPage' = 0$$

3.3 alloc_boot_pages

Allocate specified number of free pages.

Parameter #	Mode	Type	Interpretation
p0	O	unsigned long	*tFirstAllocatedPage*
p1	I	unsigned long	Number of hardware pages.
p2	I	unsigned long	Hardware page alignment.

Effects

$$\begin{aligned}&(\exists p \in HardwareMemory)(\forall \hat{p} \in HardwareMemory)(\\ &(\hat{p}.mAddress \text{ in } [p.mAddress, p.mAddress - 1] \wedge \hat{p}.cAllocated = false) \Rightarrow\\ &\hat{p}.cAllocated' = true)\end{aligned}$$

Dictionary

tFirstAllocatedPage unsigned long

$$\begin{aligned}&(\exists p \in HardwareMemory)(\forall \hat{p} \in HardwareMemory)(\\ &(\hat{p}.mAddress \text{ in } [p.mAddress, p.mAddress - 1] \wedge \hat{p}.cAllocated = false) \Rightarrow\\ &tFirstAllocatedPage' = p.mAddress)\end{aligned}$$

$$\begin{aligned}&(\nexists p \in HardwareMemory)(\forall \hat{p} \in HardwareMemory)(\\ &(\hat{p}.mAddress \text{ in } [p.mAddress, p.mAddress - 1] \wedge \hat{p}.cAllocated = false) \Rightarrow\\ &tFirstAllocatedPage' = 0)\end{aligned}$$

Issues

- Not handling hardware page alignment (parameter *p2*) correctly. Yet.
- Assume there are mMaxPage locations, so memory runs from 0 to mMaxPage - 1. *Why?*
- This effects section needs more thought. The calculation of *j* doesn't look right.

3.4 end_boot_allocator

Assign remaining free pages to domain.

Effects

$$(\forall p \in HardwareMemory)(p.mAddress \text{ in } [0, mMaxDmaPfn] \wedge p.cAllocated = false \Rightarrow p.cAllocated' = true \wedge p.cZone' = dma)$$

$$(\forall p \in HardwareMemory)(p.mAddress \text{ in } [mMaxDmaPfn + 1, mMaxPage - 1] \wedge p.cAllocated = false \Rightarrow p.cAllocated' = true \wedge p.cZone' = dom)$$

Issues

- Looks like we need to distinguish before and after state (as with the primes, above).

4 Run-Time Allocator (was *Binary Buddy Allocator*)

Secret. Xen partitions memory into a number of *zones*. The Run-Time Allocator module manages the allocation of blocks of pages of memory from, and the deallocation of blocks of pages of memory to these zones. This module hides the algorithms and data structures used to implement the allocation and deallocation of memory.

4.1 init_heap_pages

Initialize heap pages.

Parameter #	Mode	Type	Interpretation
p1	I	unsigned int	Zone.
p2	I	struct page_info *	Page.
p3	I	unsigned long	Number of pages.

Effects

$$(\forall p \in HardwareMemory)(p.mAddress \text{ in } [page2Mfn(p2), page2Mfn(p2) + p3 - 1] \Rightarrow \\ p.cAllocated' = false \wedge p.cZone' = p1)$$

4.2 free_heap_pages

Put block of pages in free space for specified zone.

Parameter #	**Mode**	**Type**	**Interpretation**
p1	I	unsigned int	Zone.
p2	I	struct page_info *	Page.
p3	I	unsigned int	*Order* of pages.

Undesired Events

- uRequestTooLarge
- uBadZone

Effects

$$(\forall p \in HardwareMemory)(p.mAddress \text{ in } [page2Mfn(p2), page2Mfn(p2) + 2^{p3} - 1] \Rightarrow p.cAllocated' = false) \wedge p.cZone' = p1)$$

Issues

- Function does not detect either UE.
- When can the zone a block belongs to change?
- I assume that the user of this function is not concerned with the merging of blocks of memory into larger blocks, nor with the algorithms and data structures involved in managing and implementing such merging.

  Of course, the *implementor* is.

4.3 alloc_heap_pages

Allocate block of pages from specified zone.

Parameter #	Mode	Type	Interpretation
p0	O	struct page_info *	*tAllocatedPageBlock*
p1	I	unsigned int	Zone.
p2	I	unsigned int	*Order* of pages.

Undesired Events

- uRequestTooLarge
- uNoSuitableBlocks

Effects

$$(\exists p \in HardwareMemory)(\forall \hat{p} \in HardwareMemory)((\hat{p}.mAddress \text{ in } [p.mAddress, p.mAddress + 2^{p2} - 1] \wedge \hat{p}.cAllocated = false \wedge \hat{p}.cZone = p1) \Rightarrow \hat{p}.cAllocated' = true)$$

Dictionary
tAllocatedPageBlock struct page_info *

$$(\exists p \in HardwareMemory)(\forall \hat{p} \in HardwareMemory)((\hat{p}.mAddress \text{ in } [p.mAddress, p.mAddress + 2^{p2} - 1] \wedge \hat{p}.cAllocated = false \wedge \hat{p}.cZone = p1) \Rightarrow tAllocatedPageBlock' = mfn2Page(p.mAddress))$$

$$(\nexists p \in HardwareMemory)(\forall \hat{p} \in HardwareMemory)((\hat{p}.mAddress \text{ in } [p.mAddress, p.mAddress + 2^{p2} - 1] \wedge \hat{p}.cAllocated = false \wedge \hat{p}.cZone = p1) \Rightarrow tAllocatedPageBlock' = null)$$

$$p2 > mMaxOrder \Rightarrow tAllocatedPageBlock' = null$$

Issues

- The function detects both UEs, but reports either by returning null.

4.4 avail_heap_pages

How many unused pages?

Parameter #	**Mode**	**Type**	**Interpretation**
p0	O	unsigned long	*tNumAvailPages*
p1	I	int	Identify one zone or all zones.

Undesired Events

- uBadZone
- uNotInitialized

Effects

None.

Dictionary

tNumAvailPages unsigned long

$$tNumAvailPages' = |\{(\forall p \in HardwareMemory)(p.cAllocated = false \wedge (p1 = p.cZone \vee p1 = -1))\}|$$

Issues

- Elsewhere, zones is declared an unsigned int. Here, zones is declared an integer, presumably to allow -1 to be used to indicate all zones.
- UEs neither detected nor reported.

4.5 scrub_heap_pages

Scrub unallocated pages from all heap zones.

Undesired Events

-

Effects

$$(\forall p \in HardwareMemory)(p.cAllocated = false \Rightarrow p.cZeroized' = true)$$

Issues

- There may be details of visible behavior this does not yet address, *e.g.*, progress dots, process pending timers.

4.6 dump_heap

Print allocation information on heap zones.

Undesired Events

-

Effects

None.

Issues

- Not capturing printouts.
- Printing is not captured in environmental model.

5 Xen Heap Allocator (was *Xen-Heap Sub-Allocator*)

Secret. The Xen Heap Allocator module manages the allocation of blocks of pages of memory from, and the deallocation of blocks of pages of memory to the xen heap zone. This module hides the algorithms and data structures used to implement the allocation and deallocation of memory.

5.1 init_xenheap_pages

Initialize xen heap pages.

Parameter #	Mode	Type	Interpretation
p1	I	paddr_t	Address of first page of xen heap.
p2	I	paddr_t	Address of last page of xen heap.

Undesired Events

- Detects but does not report $p2 \leq p1$.

Effects

$$(\forall p \in HardwareMemory)(p.mAddress \text{ in } [page2Mfn(p1), page2Mfn(p2) - 1] \Rightarrow (p.cAllocated' = false \wedge p.cZone' = xen))$$

Issues

- This doesn't yet deal with ”rounding” addresses up and down, nor with leaving one page buffer between xen and dom zones.
- Whose responsibility is it to know the location of the xen heap?

5.2 alloc_xenheap_pages

Allocate block of pages from the xen heap zone.

Parameter #	Mode	Type	Interpretation
p0	O	void *	*tAllocatedPageBlock*
p1	I	unsigned int	*Order* of pages.

Undesired Events

- uRequestTooLarge
- uNoSuitableBlocks

Effects

$$\begin{aligned}(\exists p \in HardwareMemory)(\forall \hat{p} \in HardwareMemory)(\\ (\hat{p}.mAddress \text{ in } [p.mAddress, p.mAddress + 2^{p1} - 1] \wedge\\ \hat{p}.cAllocated = false \wedge \hat{p}.cZone = xen) \Rightarrow\\ \hat{p}.cAllocated' = true)\end{aligned}$$

Dictionary

$$\begin{aligned}(\exists p \in HardwareMemory)(\forall \hat{p} \in HardwareMemory)(\\ (\hat{p}.mAddress \text{ in } [p.mAddress, p.mAddress + 2^{p1} - 1] \wedge\\ \hat{p}.cAllocated = false \wedge \hat{p}.cZone = xen) \Rightarrow\\ tAllocatedPageBlock' = mfn2Page(p.mAddress))\end{aligned}$$

$$\begin{aligned}(\nexists p \in HardwareMemory)(\forall \hat{p} \in HardwareMemory)(\\ (\hat{p}.mAddress \text{ in } [p.mAddress, p.mAddress + 2^{p1} - 1] \wedge\\ \hat{p}.cAllocated = false \wedge \hat{p}.cZone = xen) \Rightarrow\\ tAllocatedPageBlock' = null)\end{aligned}$$

$$p1 > mMaxOrder \Rightarrow tAllocatedPageBlock' = null$$

Issues

- The function detects both UEs, but reports either by returning null.

5.3 free_xenheap_pages

Put block of pages in free space for xen heap zone.

Parameter #	Mode	Type	Interpretation
p1	I	void *	Virtual address of block of pages.
p2	I	unsigned int	*Order* of pages.

Undesired Events

- uRequestTooLarge
- uNoAddress

Effects

$$
\begin{aligned}
&p1 \neq null \Rightarrow \\
&(\forall p \in HardwareMemory)((p.mAddress \text{ in } [page2Mfn(p1), page2Mfn(p1) + 2^{p2} - 1]) \Rightarrow \\
&p.cAllocated' = false) \wedge HardwarePage[i].cZone = xen)
\end{aligned}
$$

Issues

- I assume that I don't need to set all the p.cZone' = xen, since they should be already set.
- Function does not detect nor report UE.
- Can the zone a block belongs to change? So the zone should have been and should remain *xen*, eh?
- I assume that the user of this function is not concerned with the merging of blocks of memory into larger blocks, nor with the algorithms and data structures involved in managing and implementing such merging.

  Of course, the *implementor* is.

6 Dom Heap Allocator (was *Domain-Heap Sub-Allocator*)

Secret. The Dom Heap Allocator module manages the allocation of blocks of pages of memory from, and the deallocation of blocks of pages of memory to the domain heap zone. This module hides the algorithms and data structures used to implement the allocation and deallocation of memory.

6.1 init_domheap_pages

Initialize domain heap.

Parameter #	Mode	Type	Interpretation
p1	I	paddr_t	Linear address of page.
p2	I	paddr_t	Linear address of page.

Undesired Events

- Detects but does not report $p2 \leq p1$.

Effects

$$s_{dma} < e_{dma} \Rightarrow (\forall i \text{ in } [s_{dma}, e_{dma}])(HardwarePage[i].cAllocated = false \wedge HardwarePage[i].cZone = dma)$$

$$s_{dma} < e_{dma} \Rightarrow (\forall i \text{ in } [s_{dom}, e_{dom}])(HardwarePage[i].cAllocated = false \wedge HardwarePage[i].cZone = dom)$$

Dictionary

$s_{dma} = \min(p1, mMaxDmaPfn)$
$e_{dma} = \min(p2, mMaxDmaPfn)$
$s_{dom} = \max(p1, mMaxDmaPfn)$
$e_{dom} = \max(p2, mMaxDmaPfn)$

Issues

- Both init_domheap_pages and end_boot_allocator (in distinct modules) know that the dma zone goes below mMaxDmaPfn and dom zone goes above it.

  Why can't this knowledge be restricted to *one* module?
- This doesn't yet deal with "rounding" addresses up and down, nor with leaving one page buffer between xen and dom zones.
- Which module has the responsibility to know the location of the xen heap?

6.2 assign_pages

Assign pages to domain.

Parameter #	Mode	Type	Interpretation
p0	O	int	*tReturnValue*
p1	IO	struct domain *	Guest domain.
p2	IO	struct page_info []	Pages of virtual memory.
p3	I	int	Order
p4	I	int	Memory flags.

Effects

$(\forall i \text{ in } [0, 2^{p3} - 1])(\exists p \in HardwarePage)(\ (p = page2Mfn(p2[i]) \Rightarrow$

$p1.mDying$	$true$	$false$
$\neg p1.mDying$	$p1.mTotPages + 2^{p3} > p1.mMaxPages$ $\wedge \mathit{MEMF\_no\_refcount} \notin p4$	$p1.mTotPages + 2^{p3} \leq p1.mMaxPages$ $\vee \mathit{MEMF\_no\_refcount} \in p4$
$p.cPageOwner' =$	$p.cPageOwner$	$p1$
$p.cDomAllocated' =$	$p.cDomAllocated$	$true$
$p.cRefCount' =$	$p.cRefCount$	1

)

$p1.mDying$	$true$	$false$
$\neg p1.mDying$	$p1.mTotPages + 2^{p3} > p1.mMaxPages$ $\vee \mathit{MEMF\_no\_refcount} \in p4$	$p1.mTotPages + 2^{p3} \leq p1.mMaxPages$ $\wedge \mathit{MEMF\_no\_refcount} \notin p4$
$p1.cTotPages' =$	$p1.cTotPages$	$p1.cTotPages + 2^{p3}$

Dictionary

tReturnValue

$p1.mDying$	$true$	$false$
$\neg p1.mDying$	$p1.mTotPages + 2^{p3} > p1.mMaxPages$ $\wedge \mathit{MEMF\_no\_refcount} \notin p4$	$p1.mTotPages + 2^{p3} \leq p1.mMaxPages$ $\vee \mathit{MEMF\_no\_refcount} \notin p4$
tReturnValue' $=$	-1	0

Issues

- What does the call to wmb(), a macro defined in system.h, do?
- Not quite capturing `!(memflags &` *MEMF_no_refcount*`)`

6.3 __alloc_domheap_pages

Allocate heap pages for a domain.

Parameter #	Mode	Type	Interpretation
p0	O	struct page_info *	*tDomHeapPages*
p1	IO	struct domain *	Guest domain.
p2	I	unsigned int	CPU
p3	I	unsigned int	Order
p4	I	unsigned int	Memory flags.

Undesired Events

Effects

$\neg p1.mDying \wedge$ $(MEMF\_dma \notin p4 \vee$ $p1.mTotPages + 2^{p3} \leq p1.mMaxPages)$	$(\exists p \in HardwareMemory)(\forall \hat{p} \in HardwareMemory)$ ($(\hat{p}.mAddress$ in $[p.mAddress, p.mAddress + 2^{p3} - 1]$ $\wedge \hat{p}.cAllocated = false \wedge \hat{p}.cZone = dom) \Rightarrow$ $\hat{p}.cAllocated' = true)$
$\neg p1.mDying \wedge$ $(MEMF\_dma \notin p4 \vee$ $p1.mTotPages + 2^{p3} \leq p1.mMaxPages)$	$(\nexists p \in HardwareMemory)(\forall \hat{p} \in HardwareMemory)$ $(\hat{p}.mAddress$ in $[p.mAddress, p.mAddress + 2^{p3} - 1]$ $\wedge\ \hat{p}.cAllocated = false \wedge \hat{p}.cZone = dom)$ $\wedge$ $p3 \leq mMaxOrder$ $\wedge$ $tNumAvailDmaPages \geq DmaEmergencyPoolPages + 2^{p3}$ $\wedge$ $(\exists p \in HardwareMemory)(\forall \hat{p} \in HardwareMemory)$ ($(\hat{p}.mAddress$ in $[p.mAddress, p.mAddress + 2^{p3} - 1]$ $\wedge\ \hat{p}.cAllocated = false \wedge \hat{p}.cZone = dma) \Rightarrow$ $\hat{p}.cAllocated' = true)$
$\neg p1.mDying \wedge$ $MEMF\_dma \in p4 \wedge$ $p1.mTotPages + 2^{p3} \leq p1.mMaxPages$	$(\exists p \in HardwareMemory)(\forall \hat{p} \in HardwareMemory)$ ($(\hat{p}.mAddress$ in $[p.mAddress, p.mAddress + 2^{p3} - 1]$ $\wedge\ \hat{p}.cAllocated = false \wedge \hat{p}.cZone = dma) \Rightarrow$ $\hat{p}.cAllocated' = true)$

$(\forall i$ in $[0, 2^{p3} - 1])(\exists p \in HardwarePage)$ (
$(tDomHeapPage' \neq null\ \wedge\ p = page2Mfn(tDomHeapPages'[i]) \Rightarrow$

$p1.mDying$	$true$	$false$
$\neg p1.mDying$	$p1.mTotPages + 2^{p3} > p1.mMaxPages$ $\wedge\ MEMF\_no\_refcount \notin p4$	$p1.mTotPages + 2^{p3} \leq p1.mMaxPages$ $\vee\ MEMF\_no\_refcount \in p4$
$p.cPageOwner' =$	$p.cPageOwner$	$p1$
$p.cDomAllocated' =$	$p.cDomAllocated$	$true$
$p.cRefCount' =$	$p.cRefCount$	1

)

$p1.mDying$	$true$	$false$
$\neg p1.mDying$	$p1.mTotPages + 2^{p3} > p1.mMaxPages \vee$ *MEMF_no_refcount* $\in p4$	$p1.mTotPages + 2^{p3} \leq p1.mMaxPages \wedge$ *MEMF_no_refcount* $\notin p4$
$p1.cTotPages' =$	$p1.cTotPages$	$p1.cTotPages + 2^{p3}$

Dictionary

tNumAvailDmaPages unsigned long

Does this have to be redundant with the definition in `avail_heap_pages()`*?*

$$tNumAvailDmaPages = |\{(\forall p \in HardwareMemory)(p.cAllocated = false \wedge (p.cZone = dma))\}|$$

tDomHeapPages struct page_info *

$(p1 = null \vee (\neg p1.mDying \wedge MEMF\_no\_refcount \in p4 \vee p1.mTotPages + 2^{p3} \leq p1.mMaxPages)) \wedge MEMF\_dma \notin p4 \wedge p3 \leq mMaxOrder$	$(\exists p \in HardwareMemory)\ ((\forall \hat{p} \in HardwareMemory)\ (\hat{p}.mAddress \text{ in } [p.mAddress, p.mAddress + 2^{p3} - 1] \wedge \hat{p}.cAllocated = false \wedge \hat{p}.cZone = dom) \Rightarrow tDomHeapPages' = mfn2Page(p.mAddress))$
	$(\nexists p \in HardwareMemory)(\forall \hat{p} \in HardwareMemory)\ (\hat{p}.mAddress \text{ in } [p.mAddress, p.mAddress + 2^{p3} - 1] \wedge \hat{p}.cAllocated = false \wedge \hat{p}.cZone = dom) \wedge tNumAvailDmaPages \geq DmaEmergencyPoolPages + 2^{p3} \wedge (\exists p \in HardwareMemory)\ ((\forall \hat{p} \in HardwareMemory)\ (\hat{p}.mAddress \text{ in } [p.mAddress, p.mAddress + 2^{p3} - 1] \wedge \hat{p}.cAllocated = false \wedge \hat{p}.cZone = dma) \Rightarrow tDomHeapPages' = mfn2Page(p.mAddress))$
$(p1 = null \vee (\neg p1.mDying \wedge MEMF\_no\_refcount \in p4 \vee p1.mTotPages + 2^{p3} \leq p1.mMaxPages)) \wedge MEMF\_dma \in p4 \wedge p3 \leq mMaxOrder$	$(\exists p \in HardwareMemory)\ ((\forall \hat{p} \in HardwareMemory)\ (\hat{p}.mAddress \text{ in } [p.mAddress, p.mAddress + 2^{p3} - 1] \wedge \hat{p}.cAllocated = false \wedge \hat{p}.cZone = dma) \Rightarrow tDomHeapPages' = mfn2Page(p.mAddress))$
$p1.mDying \ \vee \ p3 > mMaxOrder \vee (MEMF\_no\_refcount \notin p4 \wedge p1.mTotPages + 2^{p3} \leq p1.mMaxPages) \vee (tNumAvailDmaPages < DmaEmergencyPoolPages + 2^{p3})$	$tDomHeapPages' = null$

Alternative *representation* of the value of tDomHeapPages.

$(p1 = \mathit{null} \vee$ $(\neg p1.\mathit{mDying} \wedge$ $\mathit{MEMF\_no\_refcount} \in p4 \vee$ $p1.\mathit{mTotPages} + 2^{p3} \leq$ $p1.\mathit{mMaxPages})) \wedge$ $p3 \leq \mathit{mMaxOrder}$	$\mathit{MEMF\_dma} \notin p4$	$(\exists p \in \mathit{HardwareMemory})\ ((\forall \hat{p} \in \mathit{HardwareMemory})$ $(\hat{p}.mAddress$ in $[p.mAddress, p.mAddress + 2^{p3} - 1]$ $\wedge \hat{p}.\mathit{cAllocated} = false \wedge \hat{p}.cZone = \mathit{dom})$ $\Rightarrow$ $\mathit{tDomHeapPages}' = \mathit{mfn2Page}(p.\mathit{mAddress}))$
		$(\nexists p \in \mathit{HardwareMemory})(\forall \hat{p} \in \mathit{HardwareMemory})$ $(\hat{p}.mAddress$ in $[p.mAddress, p.mAddress + 2^{p3} - 1]$ $\wedge\ \hat{p}.\mathit{cAllocated} = false \wedge \hat{p}.cZone = \mathit{dom})$ $\wedge\ \mathit{tNumAvailDmaPages} \geq \mathit{DmaEmergencyPoolPages} + 2^{p3}$ $\wedge\ (\exists p \in \mathit{HardwareMemory})\ ((\forall \hat{p} \in \mathit{HardwareMemory})$ $(\hat{p}.mAddress$ in $[p.mAddress, p.mAddress + 2^{p3} - 1]$ $\wedge\ \hat{p}.\mathit{cAllocated} = false \wedge \hat{p}.cZone = \mathit{dma})$ $\Rightarrow$ $\mathit{tDomHeapPages}' = \mathit{mfn2Page}(p.\mathit{mAddress}))$
	$\mathit{MEMF\_dma} \in p4$	$(\exists p \in \mathit{HardwareMemory})\ ((\forall \hat{p} \in \mathit{HardwareMemory})$ $(\hat{p}.mAddress$ in $[p.mAddress, p.mAddress + 2^{p3} - 1]$ $\wedge\ \hat{p}.\mathit{cAllocated} = false \wedge \hat{p}.cZone = \mathit{dma})$ $\Rightarrow$ $\mathit{tDomHeapPages}' = \mathit{mfn2Page}(p.\mathit{mAddress}))$
$p1 \neq \mathit{null} \wedge$ $p1.\mathit{mDying} \vee$ $p3 > \mathit{mMaxOrder} \vee$ $(\mathit{MEMF\_no\_refcount} \notin p4 \wedge$ $p1.\mathit{mTotPages} + 2^{p3} \leq$ $p1.\mathit{mMaxPages}) \vee$ $(\mathit{tNumAvailDmaPages} <$ $\mathit{DmaEmergencyPoolPages} + 2^{p3})$	*true*	$\mathit{tDomHeapPages}' = \mathit{null}$

Issues

6.4 alloc_domheap_pages

Allocate heap pages for a domain.

Parameter #	Mode	Type	Interpretation
p0	O	struct page_info *	*tDomHeapPages*
p1	IO	struct domain *	Guest domain.
p2	I	unsigned int	Order
p3	I	unsigned int	Memory flags.

Undesired Events

Effects

$\neg p1.\mathit{mDying} \wedge$ $(\mathit{MEMF\_dma} \notin p3 \vee$ $p1.\mathit{mTotPages} + 2^{p2} \leq p1.\mathit{mMaxPages})$	$(\exists p \in \mathit{HardwareMemory})(\forall \hat{p} \in HardwareMemory)$ ($(\hat{p}.mAddress$ in $[p.mAddress, p.mAddress + 2^{p2} - 1]$ $\wedge \hat{p}.cAllocated = false \wedge \hat{p}.cZone = dom) \Rightarrow$ $\hat{p}.\mathit{cAllocated}' = \mathit{true})$
$\neg p1.\mathit{mDying} \wedge$ $(\mathit{MEMF\_dma} \notin p3 \vee$ $p1.\mathit{mTotPages} + 2^{p2} \leq p1.\mathit{mMaxPages})$	$(\nexists p \in HardwareMemory)(\forall \hat{p} \in HardwareMemory)$ $(\hat{p}.mAddress$ in $[p.mAddress, p.mAddress + 2^{p2} - 1]$ $\wedge\ \hat{p}.cAllocated = false \wedge \hat{p}.cZone = dom)$ $\wedge$ $p2 \leq \mathit{mMaxOrder}$ $\wedge$ $\mathit{tNumAvailDmaPages} \geq \mathit{DmaEmergencyPoolPages} + 2^{p2}$ $\wedge$ $(\exists p \in HardwareMemory)(\forall \hat{p} \in HardwareMemory)$ ($(\hat{p}.mAddress$ in $[p.mAddress, p.mAddress + 2^{p2} - 1]$ $\wedge\ \hat{p}.cAllocated = false \wedge \hat{p}.cZone = dma) \Rightarrow$ $\hat{p}.\mathit{cAllocated}' = \mathit{true})$
$\neg p1.\mathit{mDying} \wedge$ $\mathit{MEMF\_dma} \in p3 \wedge$ $p1.\mathit{mTotPages} + 2^{p2} \leq p1.\mathit{mMaxPages}$	$(\exists p \in HardwareMemory)(\forall \hat{p} \in HardwareMemory)$ ($(\hat{p}.mAddress$ in $[p.mAddress, p.mAddress + 2^{p2} - 1]$ $\wedge\ \hat{p}.cAllocated = false \wedge \hat{p}.cZone = dma) \Rightarrow$ $\hat{p}.\mathit{cAllocated}' = \mathit{true})$

$(\forall i$ in $[0, 2^{p2} - 1])(\exists p \in HardwarePage)$ (
$(tDomHeapPage' \neq \mathit{null}\ \wedge\ p = \mathit{page2Mfn}(\mathit{tDomHeapPages}'[i]) \Rightarrow$

$$\left[\begin{array}{r||l|l}
p1.mDying & true & false \\
\hline
\neg p1.mDying & p1.mTotPages + 2^{p2} > p1.mMaxPages & p1.mTotPages + 2^{p2} \leq p1.mMaxPages \\
 & \wedge\ \mathit{MEMF\_no\_refcount} \notin p3 & \vee\ \mathit{MEMF\_no\_refcount} \in p3 \\
\hline\hline
p.cPageOwner' = & p.cPageOwner & p1 \\
p.cDomAllocated' = & p.cDomAllocated & true \\
p.cRefCount' = & p.cRefCount & 1
\end{array}\right]$$

)

$$\left[\begin{array}{r||l|l}
p1.mDying & true & false \\
\hline
\neg p1.mDying & p1.mTotPages + 2^{p2} > p1.mMaxPages & p1.mTotPages + 2^{p2} \leq p1.mMaxPages \\
 & \vee\ \mathit{MEMF\_no\_refcount} \in p3 & \wedge\ \mathit{MEMF\_no\_refcount} \notin p3 \\
\hline\hline
p1.cTotPages' = & p1.cTotPages & p1.cTotPages + 2^{p2}
\end{array}\right]$$

Dictionary

tNumAvailDmaPages unsigned long

Does this have to be redundant with the definition in `avail_heap_pages()`*?*

$$tNumAvailDmaPages = |\{(\forall p \in HardwareMemory)(p.cAllocated = false \wedge (p.cZone = dma))\}|$$

tDomHeapPages struct page_info *

$(p1 = null \vee (\neg p1.mDying \wedge MEMF\_no\_refcount \in p3 \vee p1.mTotPages + 2^{p2} \leq p1.mMaxPages)) \wedge MEMF\_dma \notin p3 \wedge p2 \leq mMaxOrder$	$(\exists p \in HardwareMemory)\ ((\forall \hat{p} \in HardwareMemory)$ $(\hat{p}.mAddress$ in $[p.mAddress, p.mAddress + 2^{p2} - 1]$ $\wedge \hat{p}.cAllocated = false \wedge \hat{p}.cZone = dom)$ $\Rightarrow$ $tDomHeapPages' = mfn2Page(p.mAddress))$
	$(\nexists p \in HardwareMemory)(\forall \hat{p} \in HardwareMemory)$ $(\hat{p}.mAddress$ in $[p.mAddress, p.mAddress + 2^{p2} - 1]$ $\wedge\ \hat{p}.cAllocated = false \wedge \hat{p}.cZone = dom)$ $\wedge\ tNumAvailDmaPages \geq DmaEmergencyPoolPages + 2^{p2}$ $\wedge\ (\exists p \in HardwareMemory)\ ((\forall \hat{p} \in HardwareMemory)$ $(\hat{p}.mAddress$ in $[p.mAddress, p.mAddress + 2^{p2} - 1]$ $\wedge\ \hat{p}.cAllocated = false \wedge \hat{p}.cZone = dma)$ $\Rightarrow$ $tDomHeapPages' = mfn2Page(p.mAddress))$
$(p1 = null \vee (\neg p1.mDying \wedge MEMF\_no\_refcount \in p3 \vee p1.mTotPages + 2^{p2} \leq p1.mMaxPages)) \wedge MEMF\_dma \in p3 \wedge p2 \leq mMaxOrder$	$(\exists p \in HardwareMemory)\ ((\forall \hat{p} \in HardwareMemory)$ $(\hat{p}.mAddress$ in $[p.mAddress, p.mAddress + 2^{p2} - 1]$ $\wedge\ \hat{p}.cAllocated = false \wedge \hat{p}.cZone = dma)$ $\Rightarrow$ $tDomHeapPages' = mfn2Page(p.mAddress))$
$p1.mDying\ \vee\ p2 > mMaxOrder \vee (MEMF\_no\_refcount \notin p3 \wedge p1.mTotPages + 2^{p2} \leq p1.mMaxPages) \vee (tNumAvailDmaPages < DmaEmergencyPoolPages + 2^{p2})$	$tDomHeapPages' = null$

Alternative *representation* of the value of tDomHeapPages.

$(p1 = \mathit{null} \vee$ $(\neg p1.\mathit{mDying} \wedge$ $\mathit{MEMF\_no\_refcount} \in p3 \vee$ $p1.\mathit{mTotPages} + 2^{p2} \leq$ $p1.\mathit{mMaxPages})) \wedge$ $p2 \leq \mathit{mMaxOrder}$	$\mathit{MEMF\_dma} \notin p3$	$(\exists p \in \mathit{HardwareMemory})\ ((\forall \hat{p} \in \mathit{HardwareMemory})$ $(\hat{p}.mAddress$ in $[p.mAddress, p.mAddress + 2^{p2} - 1]$ $\wedge \hat{p}.\mathit{cAllocated} = false \wedge \hat{p}.cZone = dom)$ $\Rightarrow$ $\mathit{tDomHeapPages}' = \mathit{mfn2Page}(p.\mathit{mAddress}))$
		$(\nexists p \in \mathit{HardwareMemory})(\forall \hat{p} \in \mathit{HardwareMemory})$ $(\hat{p}.mAddress$ in $[p.mAddress, p.mAddress + 2^{p2} - 1]$ $\wedge\ \hat{p}.\mathit{cAllocated} = false \wedge \hat{p}.cZone = dom)$ $\wedge\ \mathit{tNumAvailDmaPages} \geq \mathit{DmaEmergencyPoolPages} + 2^{p2}$ $\wedge\ (\exists p \in \mathit{HardwareMemory})\ ((\forall \hat{p} \in \mathit{HardwareMemory})$ $(\hat{p}.mAddress$ in $[p.mAddress, p.mAddress + 2^{p2} - 1]$ $\wedge\ \hat{p}.\mathit{cAllocated} = false \wedge \hat{p}.cZone = dma)$ $\Rightarrow$ $\mathit{tDomHeapPages}' = \mathit{mfn2Page}(p.\mathit{mAddress}))$
	$\mathit{MEMF\_dma} \in p3$	$(\exists p \in \mathit{HardwareMemory})\ ((\forall \hat{p} \in \mathit{HardwareMemory})$ $(\hat{p}.mAddress$ in $[p.mAddress, p.mAddress + 2^{p2} - 1]$ $\wedge\ \hat{p}.\mathit{cAllocated} = false \wedge \hat{p}.cZone = dma)$ $\Rightarrow$ $\mathit{tDomHeapPages}' = \mathit{mfn2Page}(p.\mathit{mAddress}))$
$p1 \neq \mathit{null} \wedge$ $p1.\mathit{mDying} \vee$ $p2 > \mathit{mMaxOrder} \vee$ $(\mathit{MEMF\_no\_refcount} \notin p3 \wedge$ $p1.\mathit{mTotPages} + 2^{p2} \leq$ $p1.\mathit{mMaxPages}) \vee$ $(\mathit{tNumAvailDmaPages} <$ $\mathit{DmaEmergencyPoolPages} + 2^{p2})$	*true*	$\mathit{tDomHeapPages}' = \mathit{null}$

Issues

6.5 avail_domheap_pages

How many unused pages?

Parameter #	Mode	Type	Interpretation
p0	O	unsigned long	*tNumAvailDomHeapPages*

Undesired Events

- uBadZone
- uNotInitialized

Effects

None.

Dictionary

tNumAvailDomHeapPages unsigned long

$$tNumAvailDomHeapPages' = |\{(\forall p \in HardwareMemory)(p.cAllocated = false \wedge (p1.cZone = dom \vee p1.cZone = dma))\}|$$

Issues

- UEs neither detected nor reported.
- This specification does not address dma_emergency_pool_pages.

6.6 free_domheap_pages

Put block of domain heap pages in free space.

Parameter #	Mode	Type	Interpretation
p1	I	struct page_info *	Page.
p2	I	unsigned int	*Order* of pages.

Undesired Events

- uRequestTooLarge
- uBadZone

Effects

$$\neg((p1.cPageOwner).mDying) \Rightarrow (\forall p \in HardwareMemory)(p.mAddress \text{ in } [page2Mfn(p1), page2Mfn(p1) + 2^{p2} - 1] \Rightarrow p.cAllocated' = false))$$

$$((p1.cPageOwner).mDying) \Rightarrow (\forall p \in HardwareMemory)(p.mAddress \text{ in } [page2Mfn(p1), page2Mfn(p1) + 2^{p2} - 1] \Rightarrow p.cScrubMe' = true))$$

Issues

- free_domheap_pages() and page_scrub_softirq() share a data structure, scrub_page_list and scrub_pages, which keeps track of pages freed by dying domains which Xen needs to scrub.
- Function does not detect either UE.
- When can the zone a block belongs to change?
- I assume that the user of this function is not concerned with the merging of blocks of memory into larger blocks, nor with the algorithms and data structures involved in managing and implementing such merging.

  Of course, the *implementor* is.

7 Page Scrubbing

The Page Scrubbing module hides when pages are cleared.

7.1 page_scrub_softirq

Zeroize some pages.

Effects

$$(\forall p \in \mathit{HardwareMemory})((p.cScrubMe = true) \Rightarrow \\ (p.cScrubMe' = false \wedge p.cAllocated' = false \wedge p.cZeroized' = true))$$

Issues

- free_domheap_pages() and page_scrub_softirq() share a data structure, scrub_page_list and scrub_pages, which keeps track of pages freed by dying domains which Xen needs to scrub.

7.2 avail_scrub_pages

How many pages to zeroize?

Parameter #	Mode	Type	Interpretation
p0	O	unsigned long	*tNumPagesToScrub*

Effects

None.

Dictionary

tNumPagesToScrub unsigned long

$$tNumPagesToScrub' = |\{(\forall p \in HardwareMemory)(p.cScrubMe = true)\}|$$